ESCUELA POLITÉCNICA NACIONAL
FACULTAD DE INGENIERÍA DE SISTEMAS
CONSTRUCCIÓN Y EVOLCUCIÓN DEL SOFTWARE

Nombre: Alison Lizeth Betancourt Herrera

Tema: Aplicando Principios de Código Limpio en Proyectos Reales.

Objetivos

- Evaluar la legibilidad y estructura del código fuente mediante el análisis de repositorios públicos.
- Formular propuestas de refactorización técnica fundamentadas en los principios de Código Limpio.

Introducción

En la industria actual del desarrollo de software, la escritura de código no solo responde a la necesidad de crear aplicaciones funcionales, sino que exige un compromiso riguroso con la calidad y la mantenibilidad a largo plazo. El concepto de Código Limpio trasciende la estética de la programación, se fundamenta en reglas y prácticas diseñadas para asegurar que el software sea legible, escalable y fácil de entender por cualquier miembro de un equipo de trabajo.

El presente informe documenta el análisis realizado sobre un repositorio de código abierto de alto impacto (developer-roadmap), con el fin de contrastar la teoría de los principios de diseño con la realidad de un proyecto en producción, a través de una metodología de auditoría de código, se examinan archivos con lógica de negocio real para identificar code smells, ambigüedades semánticas y estructuras complejas que dificultan la evolución del sistema, esto no solo busca señalar deficiencias, sino proponer soluciones concretas de refactorización que eleven el estándar de calidad del proyecto analizado.

Desarrollo

La calidad interna del software es un factor determinante para su longevidad y mantenibilidad. El concepto de Código Limpio no se refiere únicamente a que el programa funcione, sino a que la lógica escrita sea evidente y fácil de mantener para cualquier miembro del equipo, según Robert C. Martin, el código es leído muchas más veces de las que es escrito, por lo que la claridad debe ser la prioridad absoluta durante el desarrollo, un código limpio debe parecer haber sido escrito por alguien a quien le importa el problema que está resolviendo [1].

Para lograr esta claridad, es fundamental aplicar reglas de diseño como el uso de nombres significativos que revelen la intención de las variables y funciones sin necesidad de comentarios adicionales, igualmente, el Principio de Responsabilidad Única establece que cada módulo o función debe tener una sola razón para cambiar, evitando así la creación de funciones monolíticas que realizan múltiples tareas simultáneamente [1].

Finalmente, el proceso de mejora continua del código se basa en la detección de Code Smells, Martin Fowler define estos olores no como errores técnicos (bugs), sino como indicadores superficiales que sugieren problemas más profundos en la estructura del diseño, identificar estos patrones, como la duplicación de código o las listas largas de parámetros, es el primer paso para aplicar técnicas de refactorización que saneen el proyecto sin alterar su comportamiento externo [2].

Lo que se realizó fue el análisis de los archivos escogidos dentro del repositorio usando los principios vistos ¿Los nombres son claros?, ¿Las funciones son cortas y hacen una sola cosa?, ¿Hay comentarios útiles o innecesarios?, ¿Hay olores de código (duplicación, funciones largas, variables ambiguas) ? Los archivos que se analizaron dentro del repositorio, que estaban dentro de la carpeta lib fueron:

- ✓ auth.js: tenía utilidades de autenticación, 10 líneas de código, por ello tenía funciones simples.

```
import { removeAuthToken } from './jwt';

export const REDIRECT_PAGE_AFTER_AUTH = 'redirect_page_after_auth';

export function logout() {
  localStorage.removeItem(REDIRECT_PAGE_AFTER_AUTH);
  removeAuthToken();

  // Reloading will automatically redirect the user if required
  window.location.href = '/';
}
```

- ✓ date,js tenía utilidades de formateo de fechas, 77 líneas de código, por ello tenía una complejidad media.

```
import dayjs from 'dayjs';

export function getRelativeTimeString(
  date: string | Date,
  isTimed: boolean = false,
): string {
  if (!Intl?.RelativeTimeFormat) {
    return date.toString();
  }

  const rtf = new Intl.RelativeTimeFormat('en', {
    numeric: 'auto',
```

```
    style: 'narrow',
  });

  const currentDate = new Date();
  const targetDate = new Date(date);
  const diffInMilliseconds = currentDate.getTime() - targetDate.getTime();

  const diffInMinutes = Math.round(diffInMilliseconds / (1000 * 60));
  const diffInHours = Math.round(diffInMilliseconds / (1000 * 60 * 60));
  const diffInDays = Math.round(diffInMilliseconds / (1000 * 60 * 60 * 24));

  let relativeTime;

  if (diffInMinutes < 60) {
    relativeTime = rtf.format(-diffInMinutes, 'minute');
  } else if (diffInHours < 24) {
    relativeTime = rtf.format(-diffInHours, 'hour');
  } else if (diffInDays < 7) {
    if (isTimed) {
      relativeTime = dayjs(date).format('ddd h:mm A');
    } else {
      relativeTime = rtf.format(-diffInDays, 'day');
    }
  } else if (diffInDays < 30) {
    relativeTime = rtf.format(-Math.round(diffInDays / 7), 'week');
  } else if (diffInDays < 365) {
    relativeTime = rtf.format(-Math.round(diffInDays / 30), 'month');
  } else {
    if (isTimed) {
      relativeTime = dayjs(date).format('MMM D, YYYY h:mm A');
    } else {
      relativeTime = dayjs(date).format('MMM D, YYYY');
    }
  }

  if (relativeTime === 'this minute') {
    return 'just now';
  }

  return relativeTime;
}

export function formatMonthDate(date: string): string {
```

```
  return new Date(date).toLocaleDateString('en-US', {
    month: 'long',
    year: 'numeric',
  });
}

export function formatActivityDate(date: string): string {
  return new Date(date).toLocaleDateString('en-US', {
    month: 'long',
    day: 'numeric',
  });
}

export function getCurrentPeriod() {
  const now = new Date();
  const hour = now.getHours();
  if (hour < 12) {
    return 'morning';
  } else if (hour < 18) {
    return 'afternoon';
  } else {
    return 'evening';
  }
}
```

¿Los nombres son claros?

Hallazgo 01: En date.ts, la variable rtf (línea 7) es una abreviatura excesiva para RelativeTimeFormat. Obliga al lector a buscar la definición para entender qué es.

Hallazgo 02: El parámetro isTimed (línea 4) en getRelativeTimeString es ambiguo. ¿Significa que la fecha tiene hora? ¿O que *debemos mostrar* la hora? Un nombre como showTime o includeTimeInFormat sería más explícito.

Hallazgo 03: En auth.ts, la constante REDIRECT_PAGE_AFTER_AUTH es clara y utiliza correctamente la convención de mayúsculas para constantes (Screaming Snake Case).

¿Las funciones son cortas y hacen una sola cosa?

La función getRelativeTimeString es demasiado larga y compleja. Mezcla tres responsabilidades distintas:

1. Cálculos matemáticos (diffInMinutes, diffInHours).
2. Lógica de negocio para decidir el rango (si son días, semanas o años).
3. Lógica de presentación (formateo con dayjs o rtf).

La función logout en auth.ts, aunque corta, realiza acciones de diferentes capas: manipula el almacenamiento local, gestiona tokens de sesión y controla la navegación del navegador (window.location).

¿Hay comentarios útiles o innecesarios?

En auth.ts, el comentario // Reloading will automatically redirect the user if required, es útil, ya que, no explica qué hace el código, sino que explica por qué (la redirección automática), aportando contexto valioso.

¿Hay olores de código?

- **Números Mágicos:** En la función getCurrentPeriod, los números 12 y 18 aparecen sin explicación, si la definición de "tarde" cambia a las 19:00, no es obvio dónde editarlo.
- **Argumentos de bandera:** El uso de isTimed genera lógica condicional anidada (if (isTimed) ... else ...) dentro de bloques que ya son condicionales, aumentando la complejidad cognitiva.
- **Cadenas Mágicas:** En la línea 48: if (relativeTime === 'this minute') se está comparando contra un texto hardcodeado (escribir un valor fijo directamente dentro del código fuente, en lugar de obtenerlo de una variable) que podría cambiar si cambia la configuración regional, rompiendo la lógica.

A continuación, se documentó las 5 mejoras que se podrían realizar ante lo analizado:

Mejora 1: Eliminar los números mágicos en getCurrentPeriod

Se ubica en date.ts, función getCurrentPeriod, el problema es el uso del 12 y 18 que carecen de contexto semántico, la solución propuesta es extraer estos valores a constantes con nombres descriptivos al inicio del archivo, esto se hace porque aumenta la mantenibilidad, es decir, permite cambiar la definición de los periodos del día en un solo lugar sin buscar números sueltos por el código.

```
const MORNING_END_HOUR = 12;
const AFTERNOON_END_HOUR = 18;

export function getCurrentPeriod() {
  const now = new Date();
  const hour = now.getHours();
  if (hour < MORNING_END_HOUR) return 'morning';
  if (hour < AFTERNOON_END_HOUR) return 'afternoon';
  return 'evening';
}
```

Mejora 2: Reducir complejidad en getRelativeTimeString

Se ubica en date.ts, el problema es que **l**a función tiene múltiples if/else if anidados y cálculos matemáticos mezclados (Math.round(diffInMilliseconds / (1000 * 60 * 60))), la solución propuesta es extraer el cálculo de la diferencia de tiempo a una función auxiliar o usar una librería que abstraiga la matemática, ya que esto ayudará a cumplir el Principio de Responsabilidad Única, separando el cálculo matemático de la lógica de visualización, esto hace el código más robusto y menos propenso a errores de cálculo manuales.

```
// Refactorización (Simplificada usando dayjs)
const getTimeDifference = (target: Date) => {
    const now = dayjs();
    return now.diff(target, 'minute'); // Delegar la matemática
};
```

Mejora 3: Eliminar Duplicación de Código en Formateo

Se ubica en date.ts, funciones formatMonthDate y formatActivityDate, el problema es que ambas funciones hacen casi lo mismo: instancian new Date() y llaman a toLocaleDateString con 'en-US', la solución propuesta es crear una función genérica, esto se hace por el principio DRY (Don't Repeat Yourself), ya que centraliza la configuración regional ('en-US'), y si mañana la app debe soportar español ('es-ES'), solo se cambia en un lugar.

```
function formatDate(date: string, options: Intl.DateTimeFormatOptions): string {
   return new Date(date).toLocaleDateString('en-US', options);
}

export const formatMonthDate = (date: string) =>
   formatDate(date, { month: 'long', year: 'numeric' });

export const formatActivityDate = (date: string) =>
   formatDate(date, { month: 'long', day: 'numeric' });
```

Mejora 4: Corregir el Magic String propenso a errores

Se ubica en date.ts, en la línea 48, el problema es que if (relativeTime === 'this minute') { return 'just now'; } es frágil, si la librería Intl cambia su texto de salida o se cambia el idioma, esta condición fallará silenciosamente, la solución propuesta es manejar el caso de en este momento basándose en la diferencia de tiempo numérica, no en el texto de salida, esto se hace porque la lógica debe depender de calores deterministas, no de cadenas de texto de la interfaz de usuario que pueden cambiar.

```
if (Math.abs(diffInMinutes) < 1)
{
   return 'just now';
}
```

Mejora 5: Desacoplar logout del objeto global window

Se ubica en auth.ts, el problema es que la línea window.location.href = '/' hace imposible probar esta función en un entorno de pruebas sin configuraciones complejas, la solución propuesta es recibir el servicio de navegación como dependencia o envolver la redirección, ya que permite verificar que la función intenta redirigir sin necesidad de tener un navegador real ejecutando el código.

```
export function logout(navigationService = window.location) {
  localStorage.removeItem(REDIRECT_PAGE_AFTER_AUTH);
  removeAuthToken();
  navigationService.href = '/';
}
```

Conclusiones

Tras analizar el repositorio developer-roadmap, me di cuenta de que incluso en proyectos grandes y populares existe deuda técnica que debemos evitar, lo más crítico que encontré fue el uso de valores hardcodeados en date.ts, como comparar texto fijo ('this minute') o usar números sin contexto (12, 18), ya que esto hace que la aplicación sea muy frágil y se rompa si cambia el idioma o la configuración. Además, vi claramente cómo el no respetar el Principio de Responsabilidad Única en funciones largas como getRelativeTimeString complica mucho la lectura, pues mezcla matemáticas con diseño, también aprendí que escribir código corto no siempre significa código limpio, nombres abreviados como rtf o parámetros confusos como isTimed solo hacen que sea más difícil entender qué hace la función sin tener que leer todo el código.

Referencias bibliográficas

[1] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, 2008.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, 2nd ed. Boston, MA: Addison-Wesley, 2018.

Enlace del repositorio analizado:

https://github.com/kamranahmedse/developer-roadmap/tree/master/src